Efficient URL Caching for World Wide Web Crawling

Andrei Z. Broder
IBM TJ Watson Research Center
19 Skyline Dr
Hawthorne, NY 10532
abroder@us.ibm.com

Marc Najork
Microsoft Research
1065 La Avenida
Mountain View, CA 94043
najork@microsoft.com

Janet L. Wiener
Hewlett Packard Labs
1501 Page Mill Road
Palo Alto, CA 94304
janet.wiener@hp.com

ABSTRACT

Crawling the web is deceptively simple: the basic algorithm is (a) Fetch a page (b) Parse it to extract all linked URLs (c) For all the URLs not seen before, repeat (a)–(c). However, the size of the web (estimated at over 4 billion pages) and its rate of change (estimated at 7% per week) move this plan from a trivial programming exercise to a serious algorithmic and system design challenge. Indeed, these two factors alone imply that for a reasonably fresh and complete crawl of the web, step (a) must be executed about a thousand times per second, and thus the membership test (c) must be done well over ten thousand times per second against a set too large to store in main memory. This requires a distributed architecture, which further complicates the membership test.

A crucial way to speed up the test is to *cache*, that is, to store in main memory a (dynamic) subset of the "seen" URLs. The main goal of this paper is to carefully investigate several URL caching techniques for web crawling. We consider both practical algorithms: random replacement, static cache, LRU, and CLOCK, and theoretical limits: clairvoyant caching and infinite cache. We performed about 1,800 simulations using these algorithms with various cache sizes, using actual log data extracted from a massive 33 day web crawl that issued over one billion HTTP requests.

Our main conclusion is that caching is very effective – in our setup, a cache of roughly 50,000 entries can achieve a hit rate of almost 80%. Interestingly, this cache size falls at a critical point: a substantially smaller cache is much less effective while a substantially larger cache brings little additional benefit. We conjecture that such critical points are inherent to our problem and venture an explanation for this phenomenon.

Categories and Subject Descriptors

H.4.m [**Information Systems**]: Miscellaneous; D.2 [**Software**]: Software Engineering; D.2.8 [**Software Engineering**]: Metrics; D.2.11 [**Software Engineering**]: Software Architectures; H.5.4 [**Information Interfaces and Presentation**]: Hypertext/ Hypermedia—*Navigation*

General Terms

Algorithms, Measurement, Experimentation, Performance

Keywords

Caching, Crawling, Distributed crawlers, URL caching, Web graph models, Web crawlers

1. INTRODUCTION

A recent Pew Foundation study [31] states that "[s]earch engines have become an indispensable utility for Internet users" and estimates that as of mid-2002, slightly over 50% of all Americans have used web search to find information. Hence, the technology that powers web search is of enormous practical interest. In this paper, we concentrate on one aspect of the search technology, namely the process of collecting web pages that eventually constitute the search engine corpus.

Search engines collect pages in many ways, among them direct URL submission, paid inclusion, and URL extraction from nonweb sources, but the bulk of the corpus is obtained by recursively exploring the web, a process known as *crawling* or *spidering*. The basic algorithm is

(a) Fetch a page
(b) Parse it to extract all linked URLs
(c) For all the URLs not seen before, repeat (a)–(c)

Crawling typically starts from a set of *seed* URLs, made up of URLs obtained by other means as described above and/or made up of URLs collected during previous crawls. Sometimes crawls are started from a single well connected page, or a directory such as `yahoo.com`, but in this case a relatively large portion of the web (estimated at over 20%) is never reached. See [9] for a discussion of the graph structure of the web that leads to this phenomenon.

If we view web pages as nodes in a graph, and hyperlinks as directed edges among these nodes, then crawling becomes a process known in mathematical circles as *graph traversal*. Various strategies for graph traversal differ in their choice of which node among the nodes not yet explored to explore next. Two standard strategies for graph traversal are *Depth First Search (DFS)* and *Breadth First Search (BFS)* – they are easy to implement and taught in many introductory algorithms classes. (See for instance [34]).

However, crawling the web is not a trivial programming exercise but a serious algorithmic and system design challenge because of the following two factors.

1. The web is very large. Currently, Google [20] claims to have indexed over 3 billion pages. Various studies [3, 27, 28] have indicated that, historically, the web has doubled every 9-12 months.

2. Web pages are changing rapidly. If "change" means "any change", then about 40% of all web pages change weekly [12]. Even if we consider only pages that change by a third or more, about 7% of all web pages change weekly [17].

These two factors imply that to obtain a reasonably fresh and complete snapshot of the web, a search engine must crawl at least 100 million pages per day. Therefore, step (a) must be executed about 1,000 times per second, and the membership test in step (c) must be done well over ten thousand times per second, against a set of URLs that is too large to store in main memory. In addition, crawlers typically use a distributed architecture to crawl more pages in parallel, which further complicates the membership test: it is possible that the membership question can only be answered by a peer node, not locally.

A crucial way to speed up the membership test is to *cache* a (dynamic) subset of the "seen" URLs in main memory. The main goal of this paper is to investigate in depth several URL caching techniques for web crawling. We examined four practical techniques: random replacement, static cache, LRU, and CLOCK, and compared them against two theoretical limits: clairvoyant caching and infinite cache when run against a trace of a web crawl that issued over one billion HTTP requests. We found that simple caching techniques are extremely effective even at relatively small cache sizes such as 50,000 entries and show how these caches can be implemented very efficiently.

The paper is organized as follows: Section 2 discusses the various crawling solutions proposed in the literature and how caching fits in their model. Section 3 presents an introduction to caching techniques and describes several theoretical and practical algorithms for caching. We implemented these algorithms under the experimental setup described in Section 4. The results of our simulations are depicted and discussed in Section 5, and our recommendations for practical algorithms and data structures for URL caching are presented in Section 6. Section 7 contains our conclusions and directions for further research.

2. CRAWLING

Web crawlers are almost as old as the web itself, and numerous crawling systems have been described in the literature. In this section, we present a brief survey of these crawlers (in historical order) and then discuss why most of these crawlers could benefit from URL caching.

The crawler used by the Internet Archive [10] employs multiple crawling processes, each of which performs an exhaustive crawl of 64 hosts at a time. The crawling processes save non-local URLs to disk; at the end of a crawl, a batch job adds these URLs to the per-host seed sets of the next crawl.

The original Google crawler, described in [7], implements the different crawler components as different processes. A single URL server process maintains the set of URLs to download; crawling processes fetch pages; indexing processes extract words and links; and URL resolver processes convert relative into absolute URLs, which are then fed to the URL Server. The various processes communicate via the file system.

For the experiments described in this paper, we used the *Mercator* web crawler [22, 29]. Mercator uses a set of independent, communicating web crawler processes. Each crawler process is responsible for a subset of all web servers; the assignment of URLs to crawler processes is based on a hash of the URL's host component. A crawler that discovers an URL for which it is not responsible sends this URL via TCP to the crawler that is responsible for it, batching URLs together to minimize TCP overhead. We describe Mercator in more detail in Section 4.

Cho and Garcia-Molina's crawler [13] is similar to Mercator. The system is composed of multiple independent, communicating web crawler processes (called "C-procs"). Cho and Garcia-Molina consider different schemes for partitioning the URL space, including URL-based (assigning an URL to a C-proc based on a hash of the entire URL), site-based (assigning an URL to a C-proc based on a hash of the URL's host part), and hierarchical (assigning an URL to a C-proc based on some property of the URL, such as its top-level domain).

The *WebFountain* crawler [16] is also composed of a set of independent, communicating crawling processes (the "ants"). An ant that discovers an URL for which it is not responsible, sends this URL to a dedicated process (the "controller"), which forwards the URL to the appropriate ant.

UbiCrawler (formerly known as *Trovatore*) [4, 5] is again composed of multiple independent, communicating web crawler processes. It also employs a controller process which oversees the crawling processes, detects process failures, and initiates fail-over to other crawling processes.

Shkapenyuk and Suel's crawler [35] is similar to Google's; the different crawler components are implemented as different processes. A "crawling application" maintains the set of URLs to be downloaded, and schedules the order in which to download them. It sends download requests to a "crawl manager", which forwards them to a pool of "downloader" processes. The downloader processes fetch the pages and save them to an NFS-mounted file system. The crawling application reads those saved pages, extracts any links contained within them, and adds them to the set of URLs to be downloaded.

Any web crawler must maintain a collection of URLs that are to be downloaded. Moreover, since it would be unacceptable to download the same URL over and over, it must have a way to avoid adding URLs to the collection more than once. Typically, avoidance is achieved by maintaining a set of discovered URLs, covering the URLs in the frontier as well as those that have already been downloaded. If this set is too large to fit in memory (which it often is, given that there are billions of valid URLs), it is stored on disk and caching popular URLs in memory is a win: Caching allows the crawler to discard a large fraction of the URLs without having to consult the disk-based set.

Many of the distributed web crawlers described above, namely Mercator [29], WebFountain [16], UbiCrawler[4], and Cho and Molina's crawler [13], are comprised of cooperating crawling processes, each of which downloads web pages, extracts their links, and sends these links to the peer crawling process responsible for it. However, there is no need to send a URL to a peer crawling process more than once. Maintaining a cache of URLs and consulting that cache before sending a URL to a peer crawler goes a long way toward reducing transmissions to peer crawlers, as we show in the remainder of this paper.

3. CACHING

In most computer systems, memory is *hierarchical*, that is, there exist two or more levels of memory, representing different tradeoffs between size and speed. For instance, in a typical workstation there is a very small but very fast on-chip memory, a larger but slower RAM memory, and a very large and much slower disk memory. In a network environment, the hierarchy continues with network accessible storage and so on. *Caching* is the idea of storing frequently used items from a slower memory in a faster memory. In the right circumstances, caching greatly improves the performance of the overall system and hence it is a fundamental technique in the design of operating systems, discussed at length in any standard textbook [21, 37]. In the web context, caching is often mentioned in the context of a web proxy caching web pages [26, Chapter 11]. In our web crawler context, since the number of visited URLs becomes too large to store in main memory, we store the collection of visited URLs on disk, and cache a small portion in main memory.

Caching terminology is as follows: the *cache* is memory used to store equal sized atomic items. A cache has size k if it can store at most k items.[1] At each unit of time, the cache receives a *request* for an item. If the requested item is in the cache, the situation is called a *hit* and no further action is needed. Otherwise, the situation is called a *miss* or a *fault*. If the cache has fewer than k items, the missed item is added to the cache. Otherwise, the algorithm must choose either to evict an item from the cache to make room for the missed item, or not to add the missed item. The *caching policy* or *caching algorithm* decides which item to evict. The goal of the caching algorithm is to minimize the number of misses.

Clearly, the larger the cache, the easier it is to avoid misses. Therefore, the performance of a caching algorithm is characterized by the miss ratio for a given size cache.

In general, caching is successful for two reasons:

- *Non-uniformity of requests.* Some requests are much more popular than others. In our context, for instance, a link to `yahoo.com` is a much more common occurrence than a link to the authors' home pages.

- *Temporal correlation* or *locality of reference*. Current requests are more likely to duplicate requests made in the recent past than requests made long ago. The latter terminology comes from the computer memory model – data needed now is likely to be close in the address space to data recently needed. In our context, temporal correlation occurs first because links tend to be repeated on the same page – we found that on average about 30% are duplicates, cf. Section 4.2, and second, because pages on a given host tend to be explored sequentially and they tend to share many links. For example, many pages on a Computer Science department server are likely to share links to other Computer Science departments in the world, notorious papers, etc.

Because of these two factors, a cache that contains popular requests and recent requests is likely to perform better than an arbitrary cache. Caching algorithms try to capture this intuition in various ways.

We now describe some standard caching algorithms, whose performance we evaluate in Section 5.

3.1 Infinite cache (INFINITE)

This is a theoretical algorithm that assumes that the size of the cache is larger than the number of distinct requests. Clearly, in this case the number of misses is exactly the number of distinct requests, so we might as well consider this cache infinite. In any case, it is a simple bound on the performance of any algorithm.

3.2 Clairvoyant caching (MIN)

More than 35 years ago, László Belady [2] showed that if the entire sequence of requests is known in advance (in other words, the algorithm is *clairvoyant*), then the best strategy is to evict the item whose next request is farthest away in time. This theoretical algorithm is denoted MIN because it achieves the minimum number of misses on any sequence and thus it provides a tight bound on performance. Simulating MIN on a trace of the size we consider requires some care – we explain how we did it in Section 4. In Section 5 we compare the performance of the practical algorithms described below both in absolute terms and relative to MIN.

[1]The items are often called *pages* because caches were primarily used to store virtual memory pages. We shun this term to avoid confusion with web pages. Also, the requirement that all items have the same size is not essential, but simplifies the exposition, and it applies in our case.

3.3 Least recently used (LRU)

The LRU algorithm evicts the item in the cache that has not been requested for the longest time. The intuition for LRU is that an item that has not been needed for a long time in the past will likely not be needed for a long time in the future, and therefore the number of misses will be minimized in the spirit of Belady's algorithm.

Despite the admonition that "past performance is no guarantee of future results", sadly verified by the current state of the stock markets, in practice, LRU is generally very effective. However, it requires maintaining a priority queue of requests. This queue has a processing time cost and a memory cost. The latter is usually ignored in caching situations where the items are large. However, in our case, each item is only 8-10 bytes long, so the relative cost of an additional pointer per item is substantial. Fortunately, we show in Section 5 that the algorithm CLOCK, described below, performs equally well for URL caching, and its memory cost is only one extra bit per item. (See Section 6).

3.4 CLOCK

CLOCK is a popular approximation of LRU, invented in the late sixties [15]. An array of *mark* bits $M_0, M_1, \ldots, M_k$ corresponds to the items currently in the cache of size k. The array is viewed as a circle, that is, the first location follows the last. A *clock handle* points to one item in the cache. When a request X arrives, if the item X is in the cache, then its mark bit is turned on. Otherwise, the handle moves sequentially through the array, turning the mark bits off, until an unmarked location is found. The cache item corresponding to the unmarked location is evicted and replaced by X. The clock handle then begins at the next sequential location on the next cache miss.

3.5 Random replacement (RANDOM)

Random replacement (RANDOM) completely ignores the past. If the item requested is not in the cache, then a random item from the cache is evicted and replaced. This policy might seem a silly idea, but if the evicted item is chosen uniformly at random, then an item newly added to a cache of size k is expected to survive k subsequent evictions. Thus, a popular item will have the time to contribute a fair number of hits during its life in the cache, and will likely be reinstated soon. Note that the set of items in the cache at a given time is not at all a random set: recently requested items are much more likely to be in the cache than items last requested long ago.

In most practical situations, random replacement performs worse than CLOCK but not much worse. Our results exhibit a similar pattern, as we show in Section 5. RANDOM can be implemented without any extra space cost; see Section 6.

3.6 Static caching (STATIC)

If we assume that each item has a certain fixed probability of being requested, *independently* of the previous history of requests, then at any point in time the probability of a hit in a cache of size k is maximized if the cache contains the k items that have the highest probability of being requested.

There are two issues with this approach: the first is that in general these probabilities are not known in advance; the second is that the independence of requests, although mathematically appealing, is antithetical to the locality of reference present in most practical situations.

In our case, the first issue can be finessed: we might assume that the most popular k URLs discovered in a previous crawl are pretty much the k most popular URLs in the current crawl. (There are also efficient techniques for discovering the most popular items in a stream of data [18, 1, 11]. Therefore, an on-line approach might work as well.) Of course, for simulation purposes we can do a first pass over our input to determine the k most popular URLs, and then preload the cache with these URLs, which is what we did in our experiments.

The second issue above is the very reason we decided to test STATIC: if STATIC performs well, then the conclusion is that there is little locality of reference. If STATIC performs relatively poorly, then we can conclude that our data manifests substantial locality of reference, that is, successive requests are highly correlated.

3.7 Theoretical analyses

Given the practical importance of caching it is not surprising that caching algorithms have been the subject of considerable theoretical scrutiny. Two approaches have emerged for evaluating caching algorithms: *adversarial* and *average case* analysis.

Adversarial analysis compares the performance of a given on-line algorithm, such as LRU or CLOCK, against the optimal off-line algorithm MIN [6]. The ratio between their respective numbers of misses is called the competitive ratio. If this ratio is no larger than c for any possible sequence of requests then the algorithm is called c*-competitive*. If there is a sequence on which this ratio is attained, the bound is said to be *tight*. Unfortunately for the caching problem, the competitive ratio is not very interesting in practice because the worst case sequences are very artificial. For example, LRU is k-competitive, and this bound is tight. (Recall that k is the size of the cache.) For a cache of size 2^{18}, this bound tells us that LRU will never produce more than 2^{18} times the number of misses produced by MIN. However, for most cache sizes, we observed ratios for LRU of less than 1.05 (see Figure 3), and for all cache sizes, we observed ratios less than 2.

Average case analysis assumes a certain distribution on the sequence of requests. (See [23] and references therein, also [24].) The math is not easy, and the analysis usually assumes that the requests are independent, which means no locality of reference. Without locality of reference, a static cache containing the k most popular requests performs optimally on average and the issue is how much worse other algorithms do. For instance, Jelenković [23] shows that asymptotically, LRU performs at most e^{γ} (≈ 1.78; γ is Euler's constant = 0.577....) times worse than STATIC when the distribution of requests follows a Zipf law. However, such results are not very useful in our case: our requests are highly correlated, and in fact, STATIC performs worse than all the other algorithms, as illustrated in Section 5.

4. EXPERIMENTAL SETUP

We now describe the experiment we conducted to generate the crawl trace fed into our tests of the various algorithms. We conducted a large web crawl using an instrumented version of the Mercator web crawler [29]. We first describe the Mercator crawler architecture, and then report on our crawl.

4.1 Mercator crawler architecture

A Mercator crawling system consists of a number of crawling processes, usually running on separate machines. Each crawling process is responsible for a subset of all web servers, and consists of a number of worker threads (typically 500) responsible for downloading and processing pages from these servers.

Figure 1 illustrates the flow of URLs and pages through each worker thread in a system with four crawling processes.

Each worker thread repeatedly performs the following operations: it obtains a URL from the URL Frontier, which is a disk-based data structure maintaining the set of URLs to be downloaded; downloads the corresponding page using HTTP into a buffer (called a RewindInputStream or RIS for short); and, if the page is an HTML page, extracts all links from the page. The stream of extracted links is converted into absolute URLs and run through the URL Filter, which discards some URLs based on syntactic properties. For example, it discards all URLs belonging to web servers that contacted us and asked not be crawled.

The URL stream then flows into the Host Splitter, which assigns URLs to crawling processes using a hash of the URL's host name. Since most links are relative, most of the URLs (81.5% in our experiment) will be assigned to the local crawling process; the others are sent in batches via TCP to the appropriate peer crawling processes.

Both the stream of local URLs and the stream of URLs received from peer crawlers flow into the Duplicate URL Eliminator (DUE). The DUE discards URLs that have been discovered previously. The new URLs are forwarded to the URL Frontier for future download.

In order to eliminate duplicate URLs, the DUE must maintain the set of all URLs discovered so far. Given that today's web contains several billion valid URLs, the memory requirements to maintain such a set are significant. Mercator can be configured to maintain this set as a distributed in-memory hash table (where each crawling process maintains the subset of URLs assigned to it); however, this DUE implementation (which reduces URLs to 8-byte checksums, and uses the first 3 bytes of the checksum to index into the hash table) requires about 5.2 bytes per URL, meaning that it takes over 5 GB of RAM per crawling machine to maintain a set of 1 billion URLs per machine. These memory requirements are too steep in many settings, and in fact, they exceeded the hardware available to us for this experiment. Therefore, we used an alternative DUE implementation that buffers incoming URLs in memory, but keeps the bulk of URLs (or rather, their 8-byte checksums) in sorted order on disk. Whenever the in-memory buffer fills up, it is merged into the disk file (which is a very expensive operation due to disk latency) and newly discovered URLs are passed on to the Frontier.

Both the disk-based DUE and the Host Splitter benefit from URL caching. Adding a cache to the disk-based DUE makes it possible to discard incoming URLs that hit in the cache (and thus are duplicates) instead of adding them to the in-memory buffer. As a result, the in-memory buffer fills more slowly and is merged less frequently into the disk file, thereby reducing the penalty imposed by disk latency. Adding a cache to the Host Splitter makes it possible to discard incoming duplicate URLs instead of sending them to the peer node, thereby reducing the amount of network traffic. This reduction is particularly important in a scenario where the individual crawling machines are not connected via a high-speed LAN (as they were in our experiment), but are instead globally distributed. In such a setting, each crawler would be responsible for web servers "close to it".

Mercator performs an approximation of a breadth-first search traversal of the web graph. Each of the (typically 500) threads in each process operates in parallel, which introduces a certain amount of non-determinism to the traversal. More importantly, the scheduling of downloads is moderated by Mercator's *politeness policy*, which limits the load placed by the crawler on any particular web server. Mercator's politeness policy guarantees that no server ever receives multiple requests from Mercator in parallel; in addition, it guarantees that the next request to a server will only be issued after a multiple (typically 10×) of the time it took to answer the previous request has passed. Such a politeness policy is essential to any large-scale web crawler; otherwise the crawler's operator becomes inundated with complaints.

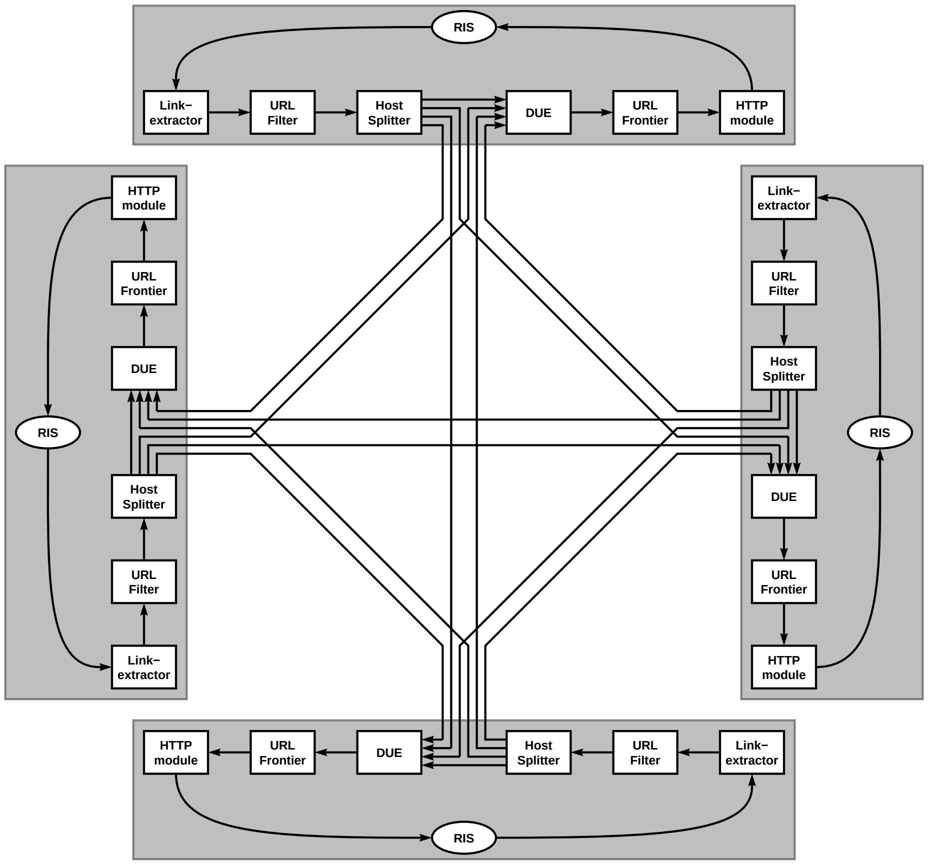

Figure 1: The flow of URLs and pages through our Mercator setup

4.2 Our web crawl

Our crawling hardware consisted of four Compaq XP1000 workstations, each one equipped with a 667 MHz Alpha processor, 1.5 GB of RAM, 144 GB of disk[2], and a 100 Mbit/sec Ethernet connection. The machines were located at the Palo Alto Internet Exchange, quite close to the Internet's backbone.

The crawl ran from July 12 until September 3, 2002, although it was actively crawling only for 33 days: the downtimes were due to various hardware and network failures. During the crawl, the four machines performed 1.04 billion download attempts, 784 million of which resulted in successful downloads. 429 million of the successfully downloaded documents were HTML pages. These pages contained about 26.83 billion links, equivalent to an average of 62.55 links per page; however, the *median* number of links per page was only 23, suggesting that the average is inflated by some pages with a very high number of links. Earlier studies reported only an average of 8 links [9] or 17 links per page [33]. We offer three explanations as to why we found more links per page. First, we configured Mercator to not limit itself to URLs found in anchor tags, but rather to extract URLs from all tags that may contain them (e.g. image tags). This configuration increases both the mean and the median number of links per page. Second, we configured it to download pages up to 16 MB in size (a setting that is significantly higher than usual), making it possible to encounter pages with tens of thousands of links. Third, most studies report the number of *unique* links per page. The numbers above include duplicate copies of a link on a page. If we only consider unique links[3] per page, then the average number of links is 42.74 and the median is 17.

The links extracted from these HTML pages, plus about 38 million HTTP redirections that were encountered during the crawl, flowed into the Host Splitter. In order to test the effectiveness of various caching algorithms, we instrumented Mercator's Host Splitter component to log all incoming URLs to disk. The Host Splitters on the four crawlers received and logged a total of 26.86 billion URLs.

After completion of the crawl, we condensed the Host Splitter logs. We hashed each URL to a 64-bit fingerprint [32, 8]. Fingerprinting is a probabilistic technique; there is a small chance that two URLs have the same fingerprint. We made sure there were no such unintentional collisions by sorting the original URL logs and counting the number of unique URLs. We then compared this

[2]144 GB of disk space was by far too small to hold both the URL log generated by the host splitter and the crawl metadata, such as the URL frontier. As the crawl progressed, we had to continuously move data to other machines with more disk space.

[3]Given that on average about 30% of the links on a page are duplicates of links on the same page, one might be tempted to eliminate these duplicates during link extraction. However, doing so is costly and has little benefit. In order to eliminate all duplicates, we have to either collect and then sort all the links on a page, or build a hash table of all the links on a page and read it out again. Given that some pages contain tens of thousands of links, such a scheme would most likely require some form of dynamic memory management. Alternatively, we could eliminate most duplicates by maintaining a fixed-sized cache of URLs that are popular on that page, but such a cache would just lower the hit-rate of the global cache while inflating the overall memory requirements.

number to the number of unique fingerprints, which we determined using an in-memory hash table on a very-large-memory machine. This data reduction step left us with four condensed host splitter logs (one per crawling machine), ranging from 51 GB to 57 GB in size and containing between 6.4 and 7.1 billion URLs.

In order to explore the effectiveness of caching with respect to inter-process communication in a distributed crawler, we also extracted a sub-trace of the Host Splitter logs that contained only those URLs that were sent to peer crawlers. These logs contained 4.92 billion URLs, or about 19.5% of all URLs. We condensed the sub-trace logs in the same fashion. We then used the condensed logs for our simulations.

5. SIMULATION RESULTS

We studied the effects of caching with respect to two streams of URLs:

1. A trace of all URLs extracted from the pages assigned to a particular machine. We refer to this as the *full trace*.
2. A trace of all URLs extracted from the pages assigned to a particular machine that were sent to one of the other machines for processing. We refer to this trace as the *cross sub-trace*, since it is a subset of the full trace.

The reason for exploring both these choices is that, depending on other architectural decisions, it might make sense to cache only the URLs to be sent to other machines or to use a separate cache just for this purpose.

We fed each trace into implementations of each of the caching algorithms described above, configured with a wide range of cache sizes. We performed about 1,800 such experiments. We first describe the algorithm implementations, and then present our simulation results.

5.1 Algorithm implementations

The implementation of each algorithm is straightforward. We use a hash table to find each item in the cache. We also keep a separate data structure of the cache items, so that we can choose one for eviction. For RANDOM, this data structure is simply a list. For CLOCK, it is a list and a clock handle, and the items also contain "mark" bits. For LRU, it is a heap, organized by last access time. STATIC needs no extra data structure, since it never evicts items. MIN is more complicated since for each item in the cache, MIN needs to know when the next request for that item will be. We therefore describe MIN in more detail.

Let A be the *trace* or sequence of requests, that is, A_t is the item requested at time t. We create a second sequence N_t containing the time when A_t next appears in A. If there is no further request for A_t after time t, we set $N_t = \infty$. Formally,

$$N_t = \begin{cases} \min\{s \mid s > t \wedge A_s = A_t\}, & \text{if } \exists s > t \text{ s.t. } A_s = A_t \\ \infty, & \text{otherwise.} \end{cases}$$

To generate the sequence N_t, we read the trace A backwards, that is, from $t_{\max}$ down to 0, and use a hash table with key A_t and value t. For each item A_t, we probe the hash table. If it is not found, we set $N_t = \infty$ and store (A_t, t) in the table. If it is found, we retrieve (A_t, t'), set $N_t = t'$, and replace (A_t, t') by (A_t, t) in the hash table.

Given N_t, implementing MIN is easy: we read A_t and N_t in parallel, and hence for each item requested, we know when it will be requested next. We tag each item in the cache with the time when it will be requested next, and if necessary, evict the item with the highest value for its next request, using a heap to identify it quickly.

5.2 Results

We present the results for only one crawling host. The results for the other three hosts are quasi-identical. Figure 2 shows the miss rate over the entire trace (that is, the percentage of misses out of all requests to the cache) as a function of the size of the cache. We look at cache sizes from $k = 2^0$ to $k = 2^{25}$. In Figure 3 we present the same data relative to the miss-rate of MIN, the optimum off-line algorithm. The same simulations for the cross-trace are depicted in Figures 4 and 5.

For both traces, LRU and CLOCK perform almost identically and only slightly worse than the ideal MIN, except in the *critical region* discussed below. RANDOM is only slightly inferior to CLOCK and LRU, while STATIC is generally much worse. Therefore, we conclude that there is considerable locality of reference in the trace, as explained in Section 3.6.

For very large caches, STATIC appears to do better than MIN. However, this is just an artifact of our accounting scheme: we only charge for misses and STATIC is not charged for the initial loading of the cache. If STATIC were instead charged k misses for the initial loading of its cache, then its miss rate would be of course worse than MIN's.

STATIC does relatively better for the cross trace than for the full trace. We believe this difference is due to the following fact: in the cross trace, we only see references from a page on one host to a page on a different host. Such references usually are short, that is, they do not go deep into a site and often they just refer to the home page. Therefore, the intersection between the most popular URLs and the cross-trace tends to be larger.

The most interesting fact is that as the cache grows, the miss rate for all of the efficient algorithms is roughly constant at 70% until the region from $k = 2^{14}$ to $k = 2^{18}$, which we call the *critical region*. Above $k = 2^{18}$, the miss rate drops abruptly to 20% after which we see only a slight improvement.

We conjecture that this critical region is due to the following phenomenon. Assume that each crawler thread h produces URLs as follows:

- With probability α, it produces URLs from a global set G, which is common to all threads.
- With probability β, it produces URLs from a local set L. This set is specific to a particular page p on a particular host h on which the thread is currently working and is further divided into $L'(p)$, the union of a small set of very popular URLs on the host h (e.g. home page, copyright notice, etc.) and a larger set of URLs already encountered[4] on the page p, and $L''(h)$, a larger set of less popular URLs on the host h.
- With probability $\gamma = 1 - \alpha - \beta$, it produces URLs chosen at random over the entire set of URLs.

We know that α is small because STATIC does not perform well. We also know that γ is about 20% for the full trace because the miss rate does not decrease substantially after the critical region. Therefore, β is large and the key factor is the interplay between L' and L''. We conjecture that for small caches, all or most of L' plus maybe the most popular fraction of G is in the cache. These URLs account for 15-30% of the hits. If L'' is substantially larger than L', say ten times larger, increasing the cache has little benefit

[4]We are grateful to Torsten Suel for observing that L' is dominated by intra-page duplicate links [36].

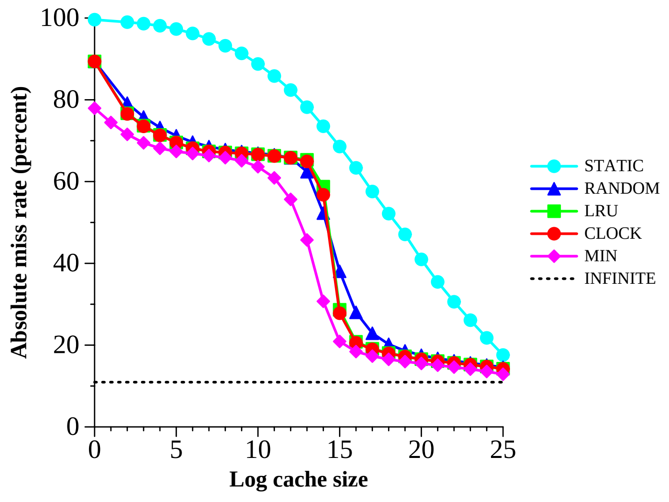

Figure 2: Miss rate as a function of cache size for the full trace

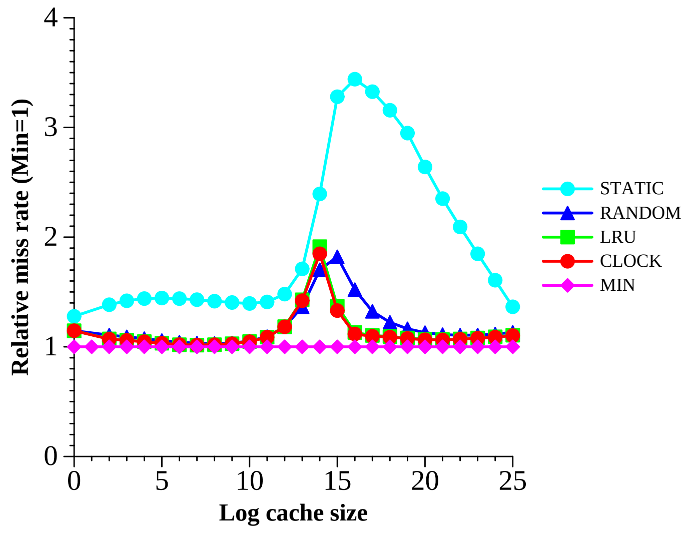

Figure 3: Relative miss rate (MIN = 1) as a function of cache size for the full trace

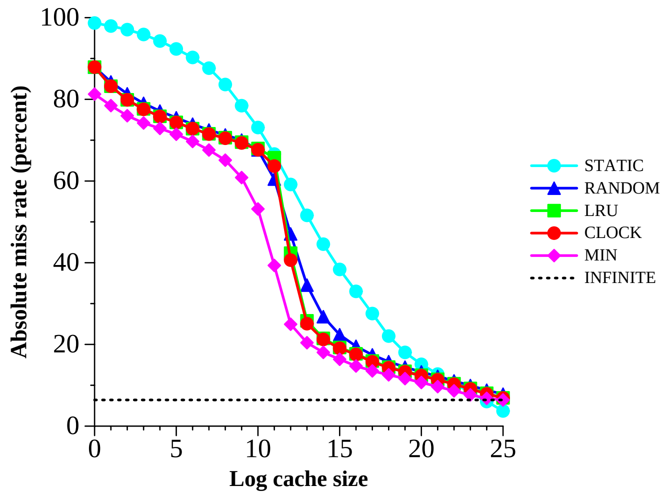

Figure 4: Miss rate as a function of cache size for the cross sub-trace

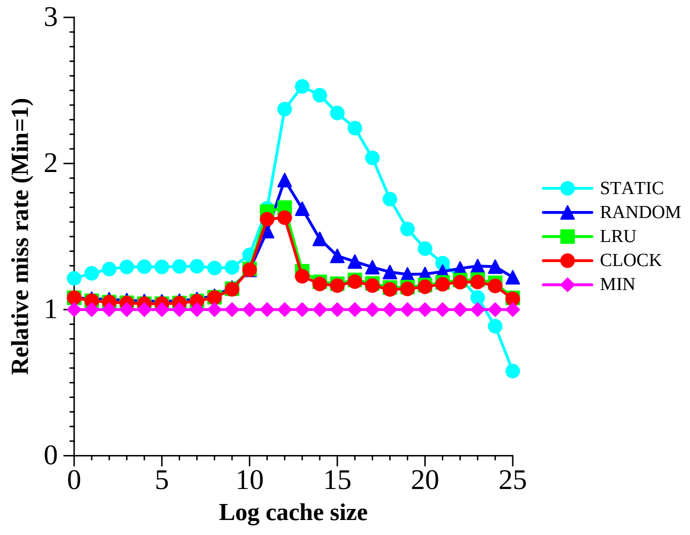

Figure 5: Relative miss rate (MIN = 1) as a function of cache size for the cross sub-trace

until a substantial portion of L'' fits in the cache. Then the increase will be linear until all of L'' is contained in the cache. After L fits entirely in the cache, we have essentially exhausted the benefits of caching, and increasing the cache size further is fruitless.

To test this hypothesis further, we performed two additional short crawls: a half-day crawl using 100 threads per process and a two-day crawl using 20 threads per process. Each crawl collected URL traces from the Host Splitter of 53–75 million URLs, which we condensed in the same manner as described earlier. We also isolated the first 70 million URLs of the full crawl's trace, which used 500 threads. Figures 6 and 7 compare the effectiveness of applying CLOCK with various cache sizes to these three traces. As expected, the critical region starts further to the left as the number of threads decreases, because with fewer threads L' is smaller and we start having a substantial portion of L'' in the cache earlier.

This trend is seen even more clearly in Figures 8 and 9 where we depict the miss rate as a function of the cache size divided by the number of threads. In this case, the curves corresponding to various number of threads almost coincide. Thus, for practical purposes the controlling variable is the allocated cache size per thread.

Finally, we investigated how the miss rate varies over time. We used a cache size 2^{18}, well past the critical region. As can be seen in Figures 10 and 11, the differences are small and the miss rate stabilizes about 1-2 billion URLs into the full trace. There is a slight increase towards the end in the full trace, probably due to the fact that by then the crawl is deep into various sites and most URLs have not been seen before. (The infinite cache also suffers an increase in miss rate.)

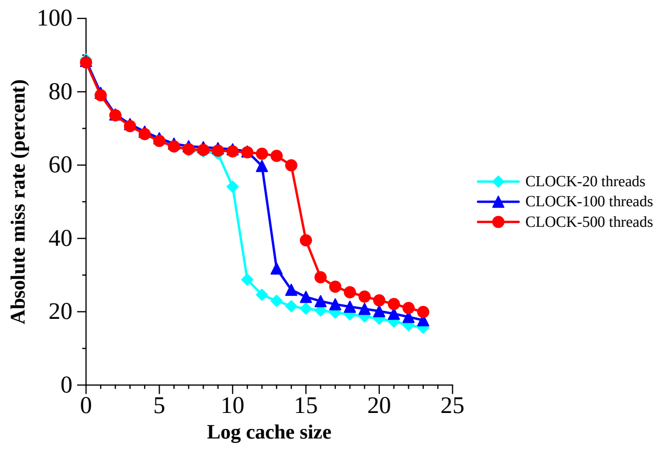

Figure 6: Miss rate as a function of cache size for the full trace for various numbers of threads (using CLOCK)

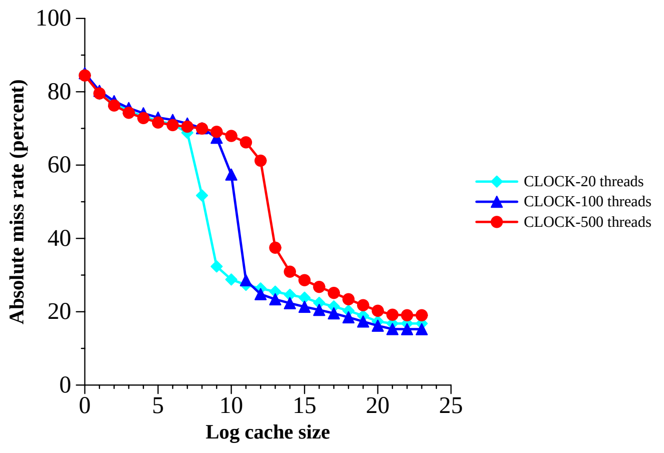

Figure 7: Miss rate as a function of cache size for the cross sub-trace for various numbers of threads (using CLOCK)

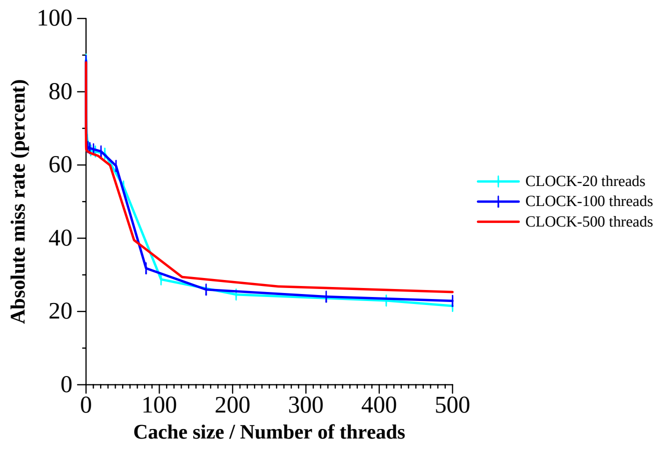

Figure 8: Miss rate as a function of cache size per thread for the full trace

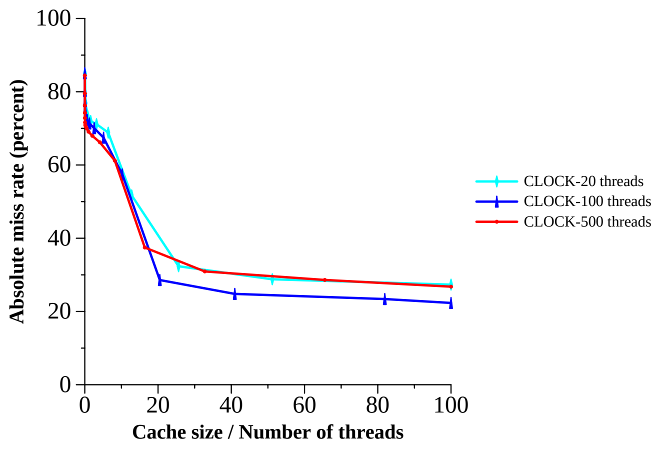

Figure 9: Miss rate as a function of cache size per thread for the cross sub-trace

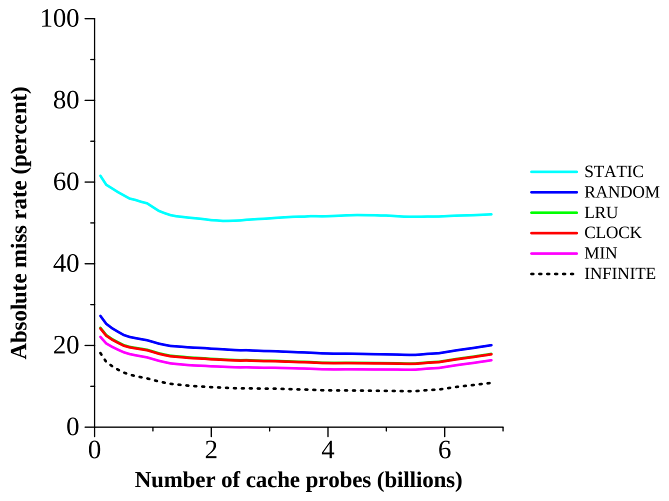

Figure 10: Miss rate as a function of time for the full trace (Cache size = 2^{18})

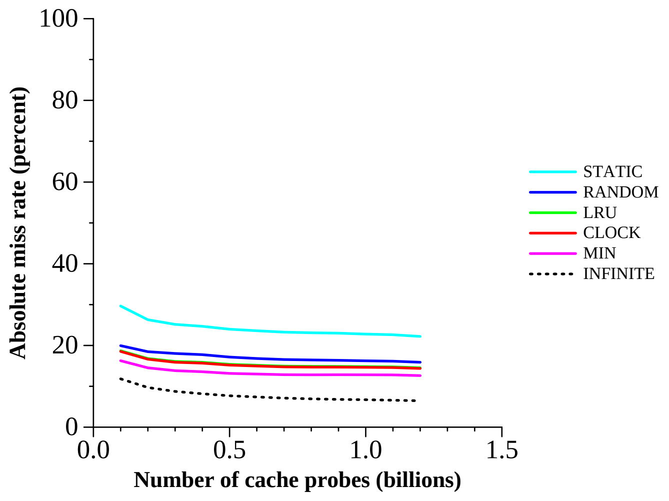

Figure 11: Miss rate as a function of time for the cross sub-trace (Cache size = 2^{18})

6. CACHE IMPLEMENTATIONS

In this section, we discuss the practical implementation of URL caching for web crawling.

To represent the cache, we need a *dictionary*, that is, a data structure that efficiently supports the basic operations *insert*, *delete* and *search*. In addition, for efficiency reasons, we would like to support the caching algorithms discussed above at minimal extra memory cost[5]. In our case, the dictionary keys are URL fingerprints, typically 8-9 bytes long. Let l be the length of this fingerprint in bits.

We present two solutions that are quite economical for both space and time: *Direct circular chaining* and *Scatter tables with circular lists*. The latter is slightly more complicated but needs only $l + 1$ bits per entry to implement the cache and RANDOM together, and $l + 2$ bits to implement the cache and CLOCK together.

For both structures, we use Lampson's abbreviated keys technique (see [25, p. 518]). The idea is to have a hash function $h(K)$ and another function $q(K)$ such that given $h(K)$ and $q(K)$, the key K can be uniquely recovered. Since in our case the keys are URL fingerprints which are already quite well distributed we can proceed as follows: To hash into a table of size 2^r we simply take the most significant r bits of the fingerprint as the hash function h and take the $l - r$ least significant bits as the function q. Thus instead of storing l bits per URL we store $l - r$ bits per URL. We will use the "saved" r bits for links among the keys that hash to the same location.

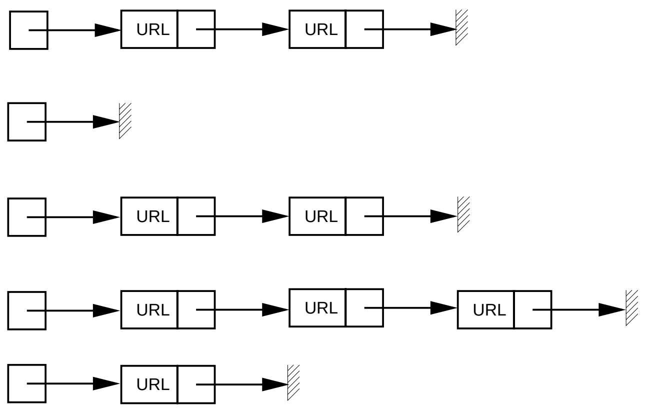

Figure 12: Hashing with direct chaining

To describe our structures it is useful to recall *hashing with direct chaining.* (See e.g. [19, section 3.3.10] and references therein). In this method, keys are hashed into a table. The keys are not stored in the table but each table entry contains a pointer to a linked list of all keys that hash to the same entry in the table as illustrated in Figure 12.

Direct circular chaining (Figure 13) is derived from direct chaining as follows: We store all the lists in one array of size $k = 2^r$, so instead of storing pointers we just store indices in the array, requiring r bits each. This array will be the array of cache entries, and RANDOM and CLOCK will be implemented on this array.

Observe that the size of the hash table can be smaller than k, say 2^{r-x}, although in this case the average length of the chains is 2^x. Using Lampson's scheme, for each entry in the array we will have to store $l - (r - x)$ bits to identify the key and r bits for the pointer, that is, a total of $l + x$ bits.

[5]The issue is not so much the extra space, but the fact that a smaller table offers faster access because a larger portion of the table fits inside a higher level of the memory hierarchy mentioned in Section 3. Caching URLs can not ignore memory caching!

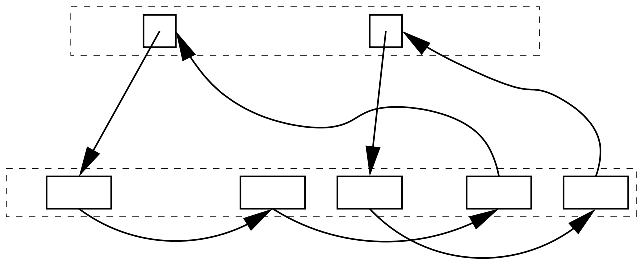

Figure 13: Hashing with direct circular chaining

But to implement RANDOM and CLOCK we need to be able to delete an arbitrary entry in the table, without knowing to which chain it belongs. To make this possible we add an extra bit per entry which indicates the last entry in the chain, and furthermore, the last entry points back to the hash table. Hence, to delete an entry we follow the chain until we get back to the hash table header and then follow the chain until we reach the entry again, but now we know its predecessor in the list and we can delete it.

The total storage needed (in bits) becomes

$$2^{r-x} \cdot r + 2^r \cdot (l + x + 1).$$

For CLOCK, we need an extra bit for the clock mark bits. For example, for a cache of size 2^{16}, using a hash table with 2^{14} entries ($x = 2$), we need a total of $l + 7$ bits per cached URL to implement RANDOM and $l + 8$ bits per URL to implement CLOCK. (We compute these numbers by dividing the total storage cost by the size of the cache.) The average chain length, which determines search and insertion time, is only 4.

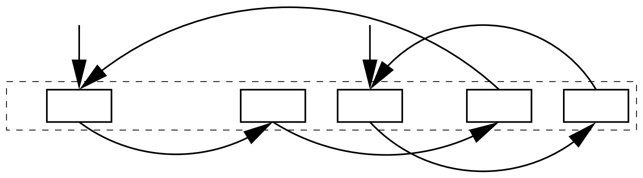

Figure 14: Scatter table with circular lists

A *scatter table with circular lists* improves on direct circular chaining by discarding the hash table entirely and hashing directly into the array that stores the keys. This data structure is the object of exercise 6.4.13 (rated 24) in Knuth's "Art of Computer Programming," Volume III [25], and we illustrate it in Figure 14.

The last element is now linked to the first element in each chain and it is more convenient for the extra bit to indicate the first element in the chain, rather than the last. Deletions are exactly as before, but insertions are more complicated: To insert an element in an empty chain starting at some location j, we first need to move the element that currently occupies location j to a free location. In the caching scenario, this is the location of the URL that has just been evicted from the cache. Although slightly more complicated, this method requires only $l + 1$ bits per cache entry to implement RANDOM and $l + 2$ to implement CLOCK.

7. CONCLUSIONS AND FUTURE DIRECTIONS

After running about 1,800 simulations over a trace containing 26.86 billion URLs, our main conclusion is that URL caching is very effective – in our setup, a cache of roughly 50,000 entries can achieve a hit rate of almost 80%. Interestingly, this size is a critical point, that is, a substantially smaller cache is ineffectual while a substantially larger cache brings little additional benefit. For practical purposes our investigation is complete: In view of our discussion in Section 5.2, we recommend a cache size of between 100 to 500 entries per crawling thread. All caching strategies perform roughly the same; we recommend using either CLOCK or RANDOM, implemented using a scatter table with circular chains. Thus, for 500 crawling threads, this cache will be about 2MB – completely insignificant compared to other data structures needed in a crawler. If the intent is only to reduce cross machine traffic in a distributed crawler, then a slightly smaller cache could be used. In either case, the goal should be to have a miss rate lower than 20%.

However, there are some open questions, worthy of further research. The first open problem is to what extent the crawl order strategy (graph traversal method) affects the caching performance. Various strategies have been proposed [14], but there are indications [30] that after a short period from the beginning of the crawl the general strategy does not matter much. Hence, we believe that caching performance will be very similar for any alternative crawling strategy. We can try to implement other strategies ourselves, but ideally we would use independent crawls. Unfortunately, crawling on web scale is not a simple endeavor, and it is unlikely that we can obtain crawl logs from commercial search engines.

In view of the observed fact that the size of the cache needed to achieve top performance depends on the number of threads, the second question is whether having a per-thread cache makes sense. In general, but not always, a global cache performs better than a collection of separate caches, because common items need to be stored only once. However, this assertion needs to be verified in the URL caching context.

The third open question concerns the explanation we propose in Section 5 regarding the scope of the links encountered on a given host. If our model is correct then it has certain implications regarding the appropriate model for the web graph, a topic of considerable interest among a wide variety of scientists: mathematicians, physicists, and computer scientists. We hope that our paper will stimulate research to estimate the cache performance under various models. Models where caching performs well due to correlation of links on a given host are probably closer to reality. We are making our URL traces available for this research by donating them to the Internet Archive.

8. ACKNOWLEDGMENTS

We would like to thank Dennis Fetterly for his help in managing our crawling machines and in fighting the various problems that we encountered during the data collection phase and Torsten Suel for his many valuable comments and suggestions.

9. REFERENCES

[1] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *Journal of Computer and System Sciences*, 58(1):137–147, 1999.

[2] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[3] K. Bharat and A. Z. Broder. A technique for measuring the relative size and overlap of public web search engines. In *Proceedings of the 7th World Wide Web Conference*, pages 379–388, 1998. `http://www7.scu.edu.au/programme/fullpapers/1937/com1937.htm`.

[4] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Trovatore: Towards a highly scalable distributed web crawler. In *Poster Proceedings of the 10th International World Wide Web Conference*, 2001. `http://www10.org/cdrom/posters/1033.pdf`.

[5] P. Boldi, B. Codenotti, M. Santini, and S. Vigna. Ubicrawler: A scalable fully distributed web crawler. In *Proceedings of the 8th Australian World Wide Web Conference*, July 2002. `http://ausweb.scu.edu.au/aw02/papers/refereed/vigna/paper.html`.

[6] A. Borodin and R. El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, 1998.

[7] S. Brin and L. Page. The anatomy of a large-scale hypertextual Web search engine. In *Proceedings of the 7th World Wide Web Conference*, pages 107–117, 1998. `http://www7.scu.edu.au/programme/fullpapers/1921/com1921.htm`.

[8] A. Z. Broder. Some applications of Rabin's fingerprinting method. In R. Capocelli, A. De Santis, and U. Vaccaro, editors, *Sequences II: Methods in Communications, Security, and Computer Science*, pages 143–152. Springer-Verlag, 1993.

[9] A. Z. Broder, R. Kumar, F. Maghoul, P. Raghavan, S. Rajagopalan, R. Stata, A. Tomkins, and J. L. Wiener. Graph structure in the Web. In *Proceedings of the 9th World Wide Web Conference*, pages 309–320, 2000. `http://www9.org/w9cdrom/160/160.html`.

[10] M. Burner. Crawling towards eternity: Building an archive of the world wide web. *Web Techniques Magazine*, 2(5), May 1997.

[11] M. Charikar, K. Chen, and M. Farach-Colton. Finding frequent items in data streams. *Lecture Notes in Computer Science*, 2380:693–703, 2002.

[12] J. Cho and H. Garcia-Molina. The evolution of the web and implications for an incremental crawler. In *Proceedings of the 26th International Conference on Very Large Databases*, 2000. `http://rose.cs.ucla.edu/~cho/papers/cho-evol.pdf`.

[13] J. Cho and H. Garcia-Molina. Parallel crawlers. In *Proceedings of the 11th International World Wide Web Conference*, 2002.

[14] J. Cho, H. Garcia-Molina, and L. Page. Efficient crawling through URL ordering. In *Proceedings of the 7th International World Wide Web Conference*, pages 161–172, Brisbane, 1998. `http://www7.scu.edu.au/programme/fullpapers/1919/com1919.htm`.

[15] F. J. Corbato. A Paging Experiment with the Multics System. Project MAC Memo MAC-M-384, Massachusetts Institute of Technology, 1968.

[16] J. Edwards, K. S. McCurley, and J. A. Tomlin. An adaptive model for optimizing performance of an incremental web crawler. In *Proceedings of the 10th International World Wide Web Conference*, pages 106–113, May 2001.

[17] D. Fetterly, M. Manasse, M. Najork, and J. L. Wiener. A large-scale study of the evolution of web pages. In *Proceedings of the 12th International World Wide Web Conference*, 2003.

[18] P. B. Gibbons and Y. Matias. Synopsis data structures for massive data sets. In *Proceedings of the 10th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 909–910. ACM Press, 1999.

[19] G. H. Gonnet and R. Baeza-Yates. *Handbook of algorithms and data structures : in Pascal and C*. Addison-Wesley, second edition, 1991.

[20] Google. `http://www.google.com`.

[21] J. L. Hennessy and D. A. Patterson. *Computer Architecture a Quantitive Approach*. Morgan Kaufmann, San Francisco, third edition, 2002.

[22] A. Heydon and M. Najork. Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219–229, 1999.

[23] P. R. Jelenković. Asymptotic approximation of the move-to-front search cost distribution and least-recently used caching fault probabilities. *Ann. Appl. Prob.*, 9(2):430–464, 1999. Available from `http://comet.ctr.columbia.edu/~predrag/mypub/mtfRevised.ps`.

[24] D. E. Knuth. An analysis of optimum caching. *Journal of Algorithms*, 6(2):181–199, 1985. Reprinted as Chapter 17 of *Selected Papers on Analysis of Algorithms* by Donald E. Knuth, Stanford, California, Center for the Study of Language and Information, 2000.

[25] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*. Addison-Wesley, 1973.

[26] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice. HTTP/1.1: Networking Protocols, Caching, and Traffic Measurement*. Addison-Wesley, May 2001.

[27] S. Lawrence and C. L. Giles. Searching the World Wide Web. *Science*, 280(5360):98–100, 1998.

[28] S. Lawrence and C. L. Giles. Accessibility of information on the web. *Nature*, 400:107–109, 1999.

[29] M. Najork and A. Heydon. High-performance web crawling. SRC Research Report 173, Compaq Systems Research Center, Palo Alto, CA, Sept. 2001.

[30] M. Najork and J. L. Wiener. Breadth-First Crawling yields high-quality pages. In *Proceedings of the 10th International World Wide Web Conference*, pages 114–118, May 2001.

[31] Pew Internet and American Life Project Survey. Search engines: A Pew Internet project data memo, July 2002. `http://www.pewinternet.org/reports/toc.asp?Report=64`.

[32] M. O. Rabin. Fingerprinting by random polynomials. Technical Report TR-15-81, Center for Research in Computing Technology, Harvard University, 1981.

[33] K. Randall, R. Stata, R. Wickremesinghe, and J. L. Wiener. The Link Database: Fast Access to Graphs of the Web. In *Proceedings of the Data Compression Conference*, pages 122–131, April, 2002.

[34] R. Sedgewick. *Algorithms in C, Part 5: Graph Algorithms*. Addison-Wesley, third edition, 2001.

[35] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *IEEE International Conference on Data Engineering (ICDE)*, Feb. 2002.

[36] T. Suel. Personal communication. Jan. 2003.

[37] A. S. Tanenbaum and A. S. Woodhull. *Operating Systems – Design and Implementation*. Prentice-Hall, second edition, 1997.